

WolfSSL

Luca Valentini

Insert Date

Contents

1	SSL Protocol	1
1.1	Introduction	1
1.2	SSL Handshake	3
2	Wolf SSL	6
3	Test case	9
3.1	Client/Server provided by WolSSL	9
3.2	EchoClient/EchoServer provided by WolfSSL	11
4	Create a program using Wolf SSL	13
4.1	TCP application	13
4.2	From TCP to SSL	18
5	Differences between WolfSSL and OpenSSL	19
5.1	Introduction	19

Abstract

Explanation of this article. Must be a synthesis

Chapter 1

SSL Protocol

1.1 Introduction

The SSL protocol is a client/server protocol that provides the following basic security services to the communicating peers:

- Authentication (both peer entity and data origin authentication) services
- Connection confidentiality services
- Connection integrity services

The SSL protocol is sockets-oriented, meaning that all or none of the data that is sent to or received from a network connection is cryptographically protected in exactly the same way. It can be best viewed as an intermediate layer between the transport and the application layer that serves two purposes:

- Establish a secure connection between the communicating peers
- Use this connection to securely transmit higher-layer protocol data from the sender to the receiver. It therefore fragments the data in pieces called fragments; each fragment is optionally compressed, authenticated, encrypted, prepended with a header, and transmitted to the receiver. Each data fragment prepared this way is sent in a distinct SSL record.

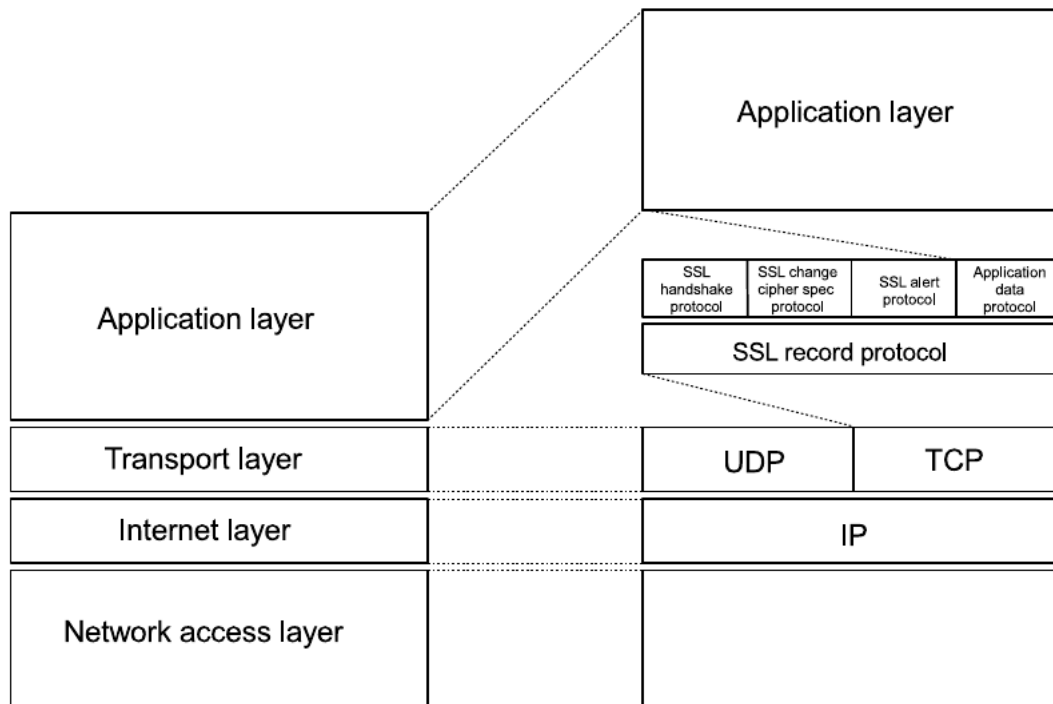


Figure 1.1: The SSL with its (sub)layer and (sub)protocols

The SSL consists of two sublayers and a few subprotocols:

- The lower sublayer is stacked on top of some connection-oriented and reliable transport layer protocol. This layer basically comprises the SSL record protocol that is used for the encapsulation of the higher-layer protocol data.
- The higher sublayer is stacked on top of the SSL record protocol and comprises four subprotocols.
 - The *SSL handshake protocol* is the core subprotocol of SSL. It is used for establishment of a secure connection. It allows the communicating peers to authenticate each other and to negotiate a cipher suite and a compression method.
 - The *SSL change cipher spec protocol* is used to put the parameters, set by the SSL handshake protocol in place and make them effective.
 - The *SSL alert protocol* allows the communicating peers to signal indicators of potential problems and send respective alert messages to each other.

- The *SSL application data protocol* is used for the secure transmission of application data.

In spite of the fact that SSL consists of several subprotocols, we use the term *SSL protocol* to refer to all of them simultaneously.

1.2 SSL Handshake

The SSL handshake protocol is layered on top of the SSL record protocol. It allows a client and server to authenticate each other and to negotiate issues like cipher suites and compression methods.

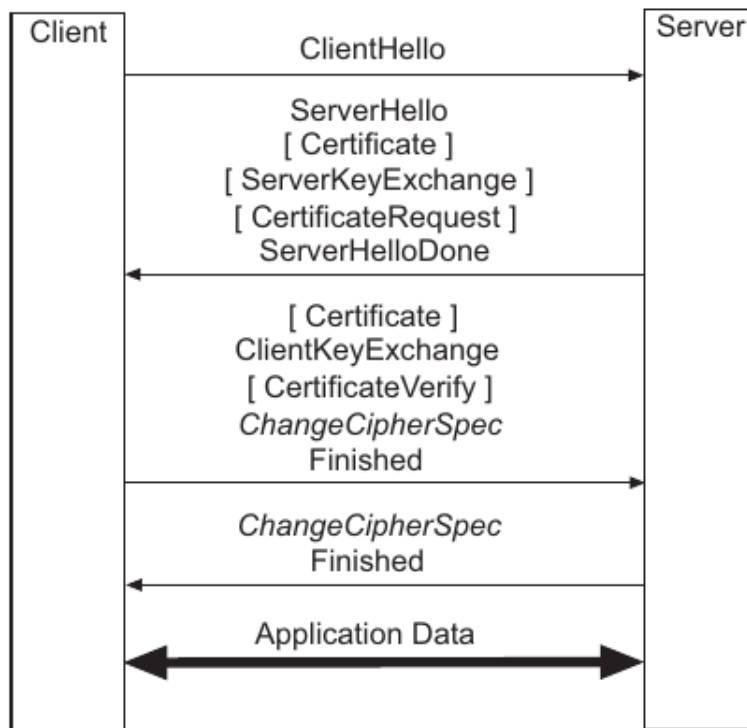


Figure 1.2: The SSL handshake protocol

The SSL handshake protocol comprises four sets of messages:

- The first flight comprises a single *ClientHello* message that is sent from the client to the server.

- The second flight comprises two messages that are sent back from the server to the client:
 1. *ServerHello* message is sent in response to the *ClientHello* message
 2. (optional) If the server is to authenticate itself, it may send a *Certificate* message to the client.
 3. (optional) Under some circumstances, the server may send a *ServerKeyExchange* message to the client.
 4. (optional) If the server requires the client to authenticate itself with a public key certificate, then it may send a *CertificateRequest* message to the client.
 5. Finally, server send a *ServerHelloDone* message to the client.
- The third flight comprises three to five messages that are again sent from the client to the server:
 1. (optional) If the server has sent a *CertificateRequest* message, then the client sends a *Certificate* message to the server.
 2. In the main step of the protocol, the client sends a *ClientKeyExchange* message to the server.
 3. (optional) If the client has sent a certificate to the server, then it must also send a *CertificateVerify* message to the server. This message is digitally signed with the private key that corresponds to the client certificate's public key.
 4. The client sends a *ChangeCipherSpec* message to the server (using the SSL change cipher spec protocol) and copies its pending write state into the current write state.
 5. The client sends a *Finished* message to the server. As mentioned above, this is the first message that is cryptographically protected under the new cipher spec.
- The fourth flight comprises two messages that are sent from the server back to the client:
 1. The server sends another *ChangeCipherSpec* message to the client and copies its pending write state into the current write state.
 2. Finally, the server send a *Finished* message to the client. Again, this message is cryptographically protected under the new cipher spec.

At this point in time, the SSL handshake is complete and the client and server may start exchanging application-layer data.

Chapter 2

Wolf SSL

The wolfSSL embedded SSL library is a lightweight SSL/TLS library written in ANSI C and targeted for embedded, RTOS, and resource-constrained environments - primarily because of its small size, speed, and feature set; It's a SSL/TLS library optimized to run on embedded platforms.

It's free and it has an excellent cross platform support.

WolfSSL supports standards up to the current TLS 1.3 and DTLS 1.2 levels, is up to 20 times smaller than OpenSSL and it's powered by the colfCrypt library.

This library is built for maximum portability and supports the C programming language as a primary interface. It also supports several other host languages, including Java (wolfSSL JNI), C# (wolfSSL C#), Python, and PHP and Perl.

To improve performance it supports hardware cryptography and acceleration on several platforms.

In the following list you can see some of WolfSSI's features:

- Runtime memory usage between 1-36 kB
- OpenSSL compatibility layer
- Hash Functions:
 - MD2
 - MD4
 - MD5
 - SHA-1
 - SHA-224
 - SHA-256
 - SHA-384
 - SHA-512
 - BLAKE2b
 - RIPEMD-160
 - Poly1305

- Mutual authentication support (client/server)
- SSL Sniffer (SSL Inspection) Support
- IPv4 and IPv6 support

The operating systems supported are:

- | | | |
|--------------------|---|---------------------------|
| 1. Win32/64 | 17. Android | 31. ARC MQX |
| 2. Linux | 18. Nintendo Wii and Gamecube through DevKitPro | 32. TI - RTOS |
| 3. Mac OS X | | 33. uTasker |
| 4. Solaris | 19. QNX | 34. embOS |
| 5. ThreadX | 20. MontaVista | 35. INtime |
| 6. VxWorks | 21. NonStop | 36. Mbed |
| 7. FreeBSD | 22. TRON / ITRON / ITRON | 37. uT - Kernel |
| 8. NetBSD | 23. Micrium C / OS - III | 38. RIOT |
| 9. OpenBSD | | 39. CMSIS -RTOS |
| 10. embedded Linux | 24. FreeRTOS | 40. FROSTED |
| 11. Yocto Linux | 25. SafeRTOS | 41. Green Hills INTEGRITY |
| 12. OpenEmbedded | 26. NXP / Freescale MQX | 42. Keil RTX |
| 13. WinCE | 27. Nucleus | 43. TOPPERS |
| 14. Haiku | 28. TinyOS | 44. PetaLinux |
| 15. OpenWRT | 29. HP / UX | 45. Apache Mynewt |
| 16. iPhone(iOS) | 30. AIX | 46. PikeOS |

Chapter 3

Test case

3.1 Client/Server provided by WolfSSL

```
server@server:~/Documents/information_system_security/wolfSSL/wolfssl/examples/server$ ./server -b
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
SSL curve name is SECP256R1
Client message: hello wolfssl!
server@server:~/Documents/information_system_security/wolfSSL/wolfssl/examples/server$
```

Figure 3.1: Server SSL

+

```
luca@luca:~/Documents/information_system_security/wolfSSL/wolfssl/examples/client$ ./client -h 192.168.0.254
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
SSL curve name is SECP256R1
I hear you fa shizzle!
luca@luca:~/Documents/information_system_security/wolfSSL/wolfssl/examples/client$
```

Figure 3.2: Client SSL

In this example, the server is a simple SSL server that allows only one client connection; after the connection with a client, the server receives an encrypted message from client, it responds and quits.

The -b parameter allows the server to bind to any interface instead of localhost only.

The client after the connection with the server, sends a message (hello wolfssl!) and after the server response, it quits.

The -h parameter allows the client to specify the server address to perform the connection.

tcp.port == 11111 && ip.addr == 192.168.0.254 && ssl					
No.	Time	Source	Destination	Protocol	Length Info
287	51.105686952	192.168.0.47	192.168.0.254	TLSv1.2	222 Client Hello
289	51.106236413	192.168.0.254	192.168.0.47	TLSv1.2	161 Server Hello
291	51.107010717	192.168.0.254	192.168.0.47	TLSv1.2	2468 Certificate
293	51.165325915	192.168.0.254	192.168.0.47	TLSv1.2	404 Server Key Exchange
295	51.165422054	192.168.0.254	192.168.0.47	TLSv1.2	98 Certificate Request, Server Hello Done
297	51.165925679	192.168.0.47	192.168.0.254	TLSv1.2	1311 Certificate
299	51.174939731	192.168.0.47	192.168.0.254	TLSv1.2	141 Client Key Exchange
301	51.189231968	192.168.0.47	192.168.0.254	TLSv1.2	335 Certificate Verify
302	51.189325871	192.168.0.47	192.168.0.254	TLSv1.2	117 Change Cipher Spec, Encrypted Handshake Message
305	51.193208227	192.168.0.254	192.168.0.47	TLSv1.2	72 Change Cipher Spec
307	51.193262294	192.168.0.254	192.168.0.47	TLSv1.2	111 Encrypted Handshake Message
311	51.193516103	192.168.0.47	192.168.0.254	TLSv1.2	109 Application Data
315	51.194124325	192.168.0.254	192.168.0.47	TLSv1.2	118 Application Data
319	51.194168545	192.168.0.254	192.168.0.47	TLSv1.2	97 Encrypted Alert
320	51.194290399	192.168.0.47	192.168.0.254	TLSv1.2	97 Encrypted Alert

▸ Frame 287: 222 bytes on wire (1776 bits), 222 bytes captured (1776 bits) on interface 0
▸ Ethernet II, Src: Dell 66:c2:8f (a4:4c:c8:66:c2:8f), Dst: AsustekC_5a:f2:0b (00:1d:60:5a:f2:0b)
▸ Internet Protocol Version 4, Src: 192.168.0.47, Dst: 192.168.0.254
▸ Transmission Control Protocol, Src Port: 41904, Dst Port: 11111, Seq: 1, Ack: 1, Len: 156
▸ Secure Sockets Layer

Figure 3.3: All SSL packets sent

Client IP: 192.168.0.47

Server IP: 192.168.0.254

Come si puo' bene vedere dalla figura soprastante, la comunicazione viene iniziata dal client con un 'Client Hello'; dopo SSL handshake, ci sono due messaggi 'Application Data' inviati rispettivamente dal client verso il server e dal server verso il client il cui contenuto e' cifrato. Una volta che il server invia la risposta al client, la comunicazione viene chiusa attraverso 'Encrypted Alert'.

▸ Frame 104: 109 bytes on wire (872 bits), 109 bytes captured (872 bits) on interface 0			
▸ Ethernet II, Src: IntelCor_f3:50:e8 (28:c6:3f:f3:50:e8), Dst: AsustekC_5a:f2:0b (00:1d:60:5a:f2:0b)			
▸ Internet Protocol Version 4, Src: 192.168.0.43, Dst: 192.168.0.254			
▸ Transmission Control Protocol, Src Port: 59580, Dst Port: 11111, Seq: 1797, Ack: 2919, Len: 43			
▸ Secure Sockets Layer			
▾ TLSv1.2 Record Layer: Application Data Protocol: Application Data Content Type: Application Data (23) Version: TLS 1.2 (0x0303) Length: 38			
Encrypted Application Data: 4e35d9cfa9e74d7042985836a47c8c531dc3275c566c64d2...			

0000	00 1d 60 5a f2 0b 28 c6 3f f3 50 e8 08 00 45 00	..`Z..(. ? .P...E.
0010	00 5f bb 6c 40 00 40 06 fc b2 c0 a8 00 2b c0 a8	.._l@.@.+...
0020	00 fe e8 bc 2b 67 dc b2 df 19 cc 79 0f 3e 80 18+g... ..y.>...
0030	01 f5 1b d0 00 00 01 01 08 0a 8a f3 1e d6 d4 aa
0040	62 9a 17 03 03 00 26 4e 35 d9 cf a9 e7 4d 70 42	b.....&N 5....MpB
0050	98 58 36 a4 7c 8c 53 1d c3 27 5c 56 6c 64 d2 18	..X6.. ..S.. '\Vld..
0060	b3 89 c6 64 d1 f1 a7 7d 09 52 e8 dc ca	...d...} ..R...

Figure 3.4: Content of the encrypted message

Come si puo' vedere, lo scambio di messaggi e' cifrato.

3.2 EchoClient/EchoServer provided by WolfSSL

```
luca@luca:~/Documents/information_system_security/wolfSSL/wolfssl/examples/echoserver$ ./echoserver
Hi Server, I'm echoClient!
client sent quit command: shutting down!
luca@luca:~/Documents/information_system_security/wolfSSL/wolfssl/examples/echoserver$
```

Figure 3.5: EchoServer SSL

```
luca@luca:~/Documents/information_system_security/wolfSSL/wolfssl/examples/echoclient$ ./echoclient
Hi Server, I'm echoClient!
Hi Server, I'm echoClient!
quit
sending server shutdown command: quit!
luca@luca:~/Documents/information_system_security/wolfSSL/wolfssl/examples/echoclient$
```

Figure 3.6: EchoClient SSL

ssl					
No.	Time	Source	Destination	Protocol	Length Info
4	0.000056186	127.0.0.1	127.0.0.1	TLSv1.2	222 Client Hello
6	0.000147786	127.0.0.1	127.0.0.1	TLSv1.2	161 Server Hello
8	0.000189939	127.0.0.1	127.0.0.1	TLSv1.2	933 Certificate
10	0.002556695	127.0.0.1	127.0.0.1	TLSv1.2	219 Server Key Exchange
12	0.002565669	127.0.0.1	127.0.0.1	TLSv1.2	75 Server Hello Done
14	0.005584226	127.0.0.1	127.0.0.1	TLSv1.2	141 Client Key Exchange
16	0.005624458	127.0.0.1	127.0.0.1	TLSv1.2	72 Change Cipher Spec
18	0.005649888	127.0.0.1	127.0.0.1	TLSv1.2	111 Encrypted Handshake Message
20	0.006849431	127.0.0.1	127.0.0.1	TLSv1.2	72 Change Cipher Spec
22	0.006879403	127.0.0.1	127.0.0.1	TLSv1.2	111 Encrypted Handshake Message
24	20.750500681	127.0.0.1	127.0.0.1	TLSv1.2	122 Application Data
26	20.750705479	127.0.0.1	127.0.0.1	TLSv1.2	122 Application Data
28	26.744191080	127.0.0.1	127.0.0.1	TLSv1.2	100 Application Data
30	26.744218051	127.0.0.1	127.0.0.1	TLSv1.2	97 Encrypted Alert
33	26.744270747	127.0.0.1	127.0.0.1	TLSv1.2	97 Encrypted Alert

<ul style="list-style-type: none"> Transmission Control Protocol, Src Port: 55864, Dst Port: 11111, Seq: 1, Ack: 1, Len: 156 Secure Sockets Layer <ul style="list-style-type: none"> TLSv1.2 Record Layer: Handshake Protocol: Client Hello <ul style="list-style-type: none"> Content Type: Handshake (22) Version: TLS 1.2 (0x0303) Length: 151 Handshake Protocol: Client Hello <ul style="list-style-type: none"> Handshake Type: Client Hello (1) Length: 147 Version: TLS 1.2 (0x0303) Random: a1baabf21e069b1336cc31bf867416f6ef906f7eec015dbc... Session ID Length: 0 Cipher Suites Length: 48
--

Figure 3.7: All SSL packets sent

Chapter 4

Create a program using Wolf SSL

4.1 TCP application

To create a SSL program you can modify your TCP program added several SSL functions.

In my example, the TCP server after configuring the socket and connecting it with the client, it creates a ClientHandler thread that launches a thread for read and a thread for write from socket. This application creates a chat between a client and a server.

```
int main()
{
    /* Create a socket that uses an internet IPv4 address,
     * Sets the socket to be stream based (TCP),
     * 0 means choose the default protocol. */
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        fprintf(stderr, "ERROR: failed to create the socket\n");
        return -1;
    }

    /* Initialize the server address struct with zeros */
    memset(&servAddr, 0, sizeof(servAddr));

    /* Fill in the server address */
    servAddr.sin_family = AF_INET;          /* using IPv4 */
    servAddr.sin_port = htons(DEFAULT_PORT); /* on DEFAULT_PORT */
}
```



```

servAddr.sin_addr.s_addr = INADDR_ANY; /* from anywhere */

/* Bind the server socket to our port */
if (bind(sockfd, (struct sockaddr *)&servAddr,
    sizeof(servAddr)) == -1)
{
    fprintf(stderr, "ERROR: failed to bind\n");
    return -1;
}

/* Listen for a new connection, allow 5 pending connections */
if (listen(sockfd, 5) == -1)
{
    fprintf(stderr, "ERROR: failed to listen\n");
    return -1;
}

/* Continue to accept clients until shutdown is issued */
while (1)
{
    printf("Waiting for a connection...\n");

    /* Accept client connections */
    if ((connd = accept(sockfd, (struct sockaddr *)&clientAddr,
        &size)) == -1)
    {
        fprintf(stderr, "ERROR: failed to accept the
            connection\n\n");
        ncurses_end();
        return -1;
    }
    pthread_t mainThread;
    pthread_create(&mainThread, NULL, ClientHandler, NULL);
    pthread_join(mainThread, NULL);
    printText("Communication is ended!\n", "System");

    if (is_end)
        break;
}
ncurses_end();
printf("Shutdown complete\n");

/* Cleanup after this connection */

```

```

    close(connd);    /* Close the connection to the client */
    /* Cleanup and return */
    close(sockfd);   /* Close the socket listening for clients
        */
    return 0;        /* Return reporting a success          */
}

```

Listing 4.1: int main()

As stated above, the clientHandler thread creates two thread, but before it waits the client username.

```

void *ClientHandler(void *args)
{
    int ret;
    /****** USERNAME */
    /* Read the client username into our buff array */
    XMEMSET(buff, 0, sizeof(buff));
    ret = read(connd, buff, sizeof(buff));
    ncurses_start();
    clearWin();
    if (ret > 0)
    {
        /* Print to stdout any data the client sends */
        strcpy(username, buff);
        char text[256];
        sprintf(text, "Client %s connected successfully", username);
        printText(text, "System");
        printText("*****\n", "System");
        fflush(stdout);
    }
    else
    {
        printText("ERROR!!", "System");
        close(sockfd);    /* Close the connection to the server */
        pthread_exit(NULL); /* End the thread execution          */
    }
    /****** */
    XMEMSET(buff, 0, sizeof(buff));

    if (pthread_create(&Treader, NULL, readBuffer, NULL))
    {
        fprintf(stderr, "Error creating thread\n");
        fflush(stdout);
    }
}

```

```

        return NULL;
    }

    if (pthread_create(&Twriter, NULL, writeBuffer, NULL))
    {
        fprintf(stderr, "Error creating thread\n");
        fflush(stdout);
        return NULL;
    }
    pthread_join(Treader, NULL);
    pthread_join(Twriter, NULL);
    /* Cleanup after this connection */
    close(connd); /* Close the connection to the client */
    pthread_exit(NULL); /* End thread execution */
}

```

Listing 4.2: clientHandler thread

ReadBuffer is a thread that allows to read messages sent from client. It has an infinite loop that read data from socket open previously; Once it gets the message, with the printText function, the message is printed to the terminal using ncurses.

```

void *readBuffer(void *args)
{
    int ret;
    while (1)
    {
        /* Read the client data into our buff array */
        XMEMSET(buffReader, 0, sizeof(buffReader));
        ret = read(connd, buffReader, sizeof(buffReader));

        if (ret > 0)
        {
            if (!strcmp(buffReader, "quit"))
            {
                pthread_cancel(Twriter);
                pthread_exit(NULL); /* End threaded execution */
            }
            printText(buffReader, username);
        }
        else
        {

```

```

        printText("ERROR READ!!", "System");
        pthread_cancel(Twriter);
        pthread_exit(NULL); /* End threaded execution */
    }
}
}

```

Listing 4.3: readBuffer thread

WriteBuffer thread has an infinite loop used for write messages from server to client.

```

void *writeBuffer(void *args)
{
    int ret;
    while (1)
    {
        read_in();
        len = XSTRLEN(Rbuffer);

        /* Reply back to the client */
        ret = write(connd, Rbuffer, len);
        printText(Rbuffer, "Server");

        if (XSTRNCMP(Rbuffer, "quit", 4) == 0)
        {
            is_end = 1;
            break;
        }

        if (ret != len)
        {
            printText("ERROR!!", "System");
            break;
        }
    }
    pthread_cancel(Treader);
    return NULL;
}

```

Listing 4.4: writeBuffer thread

4.2 From TCP to SSL

Chapter 5

Differences between WolfSSL and OpenSSL

5.1 Introduction

The main differences are:

- Memory Usage
 - WolfSSL can be up to 20 times smaller than OpenSSL; The build size is between 20 and 100 KB and the runtime memory usage between 1 and 36 KB. This gives a magior advantage of integrating in smaller embedded devices.
- Hardware Crypto
 - WolfSSL has a partnership with the most MCU manufacturers which allows to be quite early in the market to support hardware acceleration on huge list of platforms.
- Portability
 - WolfSSL is more portable than OpenSSL because is made for real-time, mobile, embedded and enterprise systems.